

Writing an Adventure game with HLA

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2003-11/0364.html>

From: Randall Hyde (*randyhyde_at_earthlink.net*)

Date: 11/21/03

Date: Fri, 21 Nov 2003 17:40:14 GMT

Hi Paul,

For an "adventure" style game, there are two facilities in the HLA Standard Library that you'll definitely want to consider: the `str.tokenize` function which breaks up a string into individual words, and the HLA `table` class that makes looking up strings a breeze.

=====
Consider the `str.tokenize` (and `str.tokenize2`) function:

```
str.tokenize( src: string; var dest:dword ); @returns( "eax" );  
str.tokenize2( src: string; var dest:dword; delims:cset ); @returns( "eax" );
```

These two routines lexically scan a string and break it up into "lexemes" (words), returning an array of pointers to each of the lexemes. The only difference between the two routines is that the `tokenize` routine uses the following default set of delimiter characters:

```
{ ' ', #9, ',', '<', '>', '|', '\', '/', '-' }
```

This character set roughly corresponds to the delimiters used by the Windows Command Window interpreter or typical Linux shells. If you do not wish to use this particular set of delimiter characters, you may call `str.tokenize2` and specify the characters you're interested in.

The `tokenize` routines begin by skipping over all delimiter characters at the beginning of the string. Once they locate a non-delimiter character, they skip forward until they find the end of the string or the next delimiter character. Then they allocate storage for a new string on the heap and copy the delimited text to this new string. A pointer to the new string is stored into the `dword` array passed as the first parameter to `tokenize(2)`. This process is repeated for each lexeme found in the `src` string.

As these functions are intended for processing command lines, any quoted string (a sequence of characters surrounded by quotes or apostrophies) is treated as a single token/string by these functions. If this behavior is a problem for you, it's real easy to modify the `str.tokenize` source file to handle this issue.

Warning: the `dest` parameter should be an array of strings. This array must be large enough to hold pointers to each lexeme found in the string. In theory, there could be as many as `str.length(src)/2` lexemes in the source string.

On return from these functions, the `EAX` register will contain the number of lexemes found and processed in the `src` string (i.e., `EAX` will contain the number of valid elements in the `dest` array).

When you are done with the strings allocated on the heap, you should free them by calling `strfree`. Note that you need to call `strfree` for each active pointer stored in the `dest` array.

Here is an example of a call to the `str.tokenize` routine:

```
program tokenizeDemo;
#include( "stdio.hhf" );
#include( "string.hhf" );
#include( "memory.hhf" );

static
  strings: string[16];
  ParseMe: string := "This string contains five words";

begin tokenizeDemo;

  str.tokenize( strings, ParseMe );
  mov( 0, ebx );
  while( ebx < eax ) do

    stdout.put
    (
      "string[",
      (type uns32 ebx),
      "]=\"",
      strings[ebx*4],
      "\"",
      nl
    );
    strfree( strings[ebx*4] );
    inc( ebx );
  endwhile;

end tokenizeDemo;
```

This program produces the following output:

```
string[0]="This"  
string[1]="string"  
string[2]="contains"  
string[3]="five"  
string[4]="words"
```

=====

The HLA Standard Library provides a "table" class that lets you easily create and manipulate "associative arrays." An "associative array" is, effectively, an array whose index is a string rather than some integer quantity, e.g.,

```
lookupTable[ "somestring" ] = somevalue;
```

The cool thing about a table, of course, is that you can use the words you've extract with `str.tokenize` as indices into a lookup table to produce a small integer value that is easy to manipulate in your assembly program.

The "table" type provided by the HLA Standard Library is actually a class type (yep, you're headed into the realm of object-oriented programming here; fortunately, you don't need to know much about OOP to use tables). The table class provides the following procedures, methods, and iterators:

```
procedure table.create( HashSize:uns32 );  
method table.destroy( FreeValue:procedure );  
method table.getNode( id:string );  
method table.lookup( id:string );  
iterator table.item();
```

The create procedure is used to initialize a lookup table. The destroy method (procedure) is used to clean up a table data structure when you're done using it (in particular, this call frees all the storage that the table routines allocate internally during operation). For our purposes, we're just going to create a table of reserved words for the game, so the destroy method isn't that useful (we'll want to keep the table around until the game is done, and when the program quits, all the storage will be freed up anyway).

The item iterator, combined with HLA's foreach loop, will "iterate" over each item in the table. This probably isn't very useful for an adventure game, but it is useful in other applications. We'll ignore the item iterator here.

The `getNode` and `lookup` methods are of primary interest to us when building an adventure game. These are the methods that we'll use to fill the lookup table and lookup nouns and verbs in the table.

Both routines look up the string you pass as the parameter in the specified lookup table. The difference between the two is what happens when the lookup routines **do not** find the string in the table. In this situation, `getNode` **adds** the string to the lookup table and returns a pointer to the corresponding table entry in the EAX register. `lookup`, on the other hand, returns NULL in the EAX register if it cannot find the string in the table. Both routines return a pointer to the table entry in EAX if they find the entry in the table.

A table entry is a record of the following type:

```
tableNode:
  record

    link: pointer to tableNode;
    Value: dword;
    id: string;

  endrecord;
```

The **only** field you should mess with is "Value". You can use "Value" for any purpose you want. You must not modify the other fields of this data structure.

In the example I've attached to this post, I use the Value field to hold a small (unique) integer value to identify the word. These integer constants are defined using the HLA enumerated data type declaration:

```
type
  words:enum
  {
    illegal_c,
    goVerb_c,
    quitVerb_c,
    northNoun_c,
    southNoun_c,
    eastNoun_c,
    westNoun_c
  };
```

Specifically, note that `goVerb_c = 1`, `quitVerb_c=2`, `northNoun_c=3`, etc.

To use the table class in an HLA program, the first thing you've got to do is declare a table variable. In the example at the end of this post, I've declared the "tbl" variable in the static section as follows:

```
tbl :table; // Noun/verb table.
```

Before you can use the tbl variable in your program, you must first initialize this table object. This is done by calling the "class constructor" for the table class. By convention, most classes use the procedure name "create" for the class constructor; the table class follows this convention. The create procedure requires a single parameter that specifies the minimum size of the hash table that the tbl object will use. Generally, this value should be between two and four times the maximum number of words you intend to put into the table. This value effects the efficiency, not the correctness, of the table lookup operations. If this value is too small, your searches will take a little longer, if this value is too large, you're going to waste a little memory. For an adventure game, the speed of the lookup is irrelevant (much faster than user input). OTOH, since we've only got one lookup table, the amount of wasted space in this table isn't very important either. I just chose the value "100" out of the blue (this is probably much larger than it needs to be, but we're only taking about 400 bytes for the table...).

Here's the call to initialize tbl:

```
tbl.create(100);
```

You *must* make this call before doing anything else with tbl.

Once you've initialized the tbl object, the next step is to fill the lookup table with the words you want to recognize and their associated Value fields. In the demo that appears later, I've just supplied six words: go, quit, north, south, east, west. It should be pretty obvious how to extend this to any number of words (modify the words enum type and just add more calls to tbl.getNode). Here's a typical initialization sequence for one word:

```
// Put the word into the table and return a pointer to the  
// new table node:
```

```
tbl.getNode( "go" );
```

```
// store the associated integer constant into the Value  
// field of this node:
```

```
mov( goVerb_c, (type tableNode [eax]).Value );
```

Once you've repeated these two instructions for each word you want in the table, looking up a word in the table is a trivial process:

```
tbl.lookup( someStr );
```

This call returns a pointer to the associated table node if the string is found in the table, it returns NULL if the string is not in the table. The return value is in EAX.

I hope that this quick exposition has sparked your imagination and given you some ideas about how easy it is to lexically scan and parse an input command from the user when using HLA Standard Library routines.

As Rene (Betov) says earlier, you could:

```
"Use a decent Assembler written by some  
decent Assembly programmer"
```

The problem with Rene's approach, alas, is that you'd have to write all this code yourself. Why bother when it's already available in the HLA Standard Library waiting to be used?

Cheers,
Randy Hyde

```
/*  
*/  
/* This is a simple program that */  
/* demonstrates HLA tables. It shows */  
/* how to use an HLA table to lookup */  
/* and identify words you've stored */  
/* in a table such as the commands you'd */  
/* normal expect in an "adventure" type */  
/* game. */  
*/  
*/  
*/
```

```
program tableDemo;
```

```
#include( "stdlib.hhf" );
```

```
// Create a list of constants for each of the  
// words we're going to recognize in our  
// noun/verb list:
```

```

type
  words:enum
  {
    illegal_c,
    goVerb_c,
    quitVerb_c,
    northNoun_c,
    southNoun_c,
    eastNoun_c,
    westNoun_c
  };

// Some variables this program uses:

static
  cnt :uns32;
  s :str.strvar(256); // user Input string.
  tokens :string[128];
  tbl :table; // Noun/verb table.

  cmdTbl :string[] :=
    [
      "illegal_c",
      "goVerb_c",
      "quitVerb_c",
      "northNoun_c",
      "southNoun_c",
      "eastNoun_c",
      "westNoun_c"
    ];

// All tables must have a "destructor" that
// cleans up after the object when the object
// is destroyed. For adventure game noun/verb
// lists no action is really necessary because
// we quit after destorying the object.
// Nonetheless, you need to supply an empty
// procedure to satisfy the table class.

  procedure destroyTable; @noframe;
  begin destroyTable;
    ret();
  end destroyTable;

begin tableDemo;

  // Create the table and its corresponding hash
  // table. The "100" parameter suggests that
  // there are probably about 100 unique words
  // in this file (the exact count isn't important,
  // if this value is too small, the program

```

```

// runs a little slower; if this value is too
// large, the program wastes a little memory).

tbl.create( 100 );

// for each word in the noun/verb list, enter the
// word into the table and assign a value to the
// "Value" field that we will retrieve later:

tbl.getNode( "go" );
mov( goVerb_c, (type tableNode [eax]).Value );
tbl.getNode( "quit" );
mov( quitVerb_c, (type tableNode [eax]).Value );
tbl.getNode( "north" );
mov( northNoun_c, (type tableNode [eax]).Value );
tbl.getNode( "south" );
mov( southNoun_c, (type tableNode [eax]).Value );
tbl.getNode( "east" );
mov( eastNoun_c, (type tableNode [eax]).Value );
tbl.getNode( "west" );
mov( westNoun_c, (type tableNode [eax]).Value );

// Okay, now demonstrate how to use this table:

begin mainloop; forever

    stdout.put( "Enter a cmd: " );
    stdin.gets( s );
    str.tokenize( s, tokens );
    mov( eax, cnt );
    for( mov( 0, ecx ); ecx < cnt; inc( ecx ) ) do

        tbl.lookup( tokens[ ecx*4 ] );
        if( eax <> NULL ) then

            mov( (type tableNode[eax]).Value, ebx );
            stdout.put
            (
                "Command value: ",
                (type uns32 ebx),
                " cmd:",
                cmdTbl[ ebx*4 ],
                nl
            );
            exitif( ebx == quitVerb_c ) mainloop;

        else

            stdout.put( "illegal word!" nl );

        endif;
    endfor;
endbegin;

```

alt.lang.asm: Writing an Adventure game with HLA

```
// Be nice, free the storage associated with
// the current word. We won't free the last
// line input, but that's okay 'cause the
// pgm is about to quit and all storage
// is freed then, anyway.

strfree( tokens[ ecx*4 ] );

endfor;

endfor; end mainloop;

// The destructor calls destroyTable for each node it frees.
// This displays the frequency and text associated with
// each word entered into the table.

tbl.destroy( &destroyTable );

end tableDemo;
```